Docket No.: 5166.P014
Express Mail No.: EL651890191US

UNITED STATES PATENT APPLICATION

FOR

SYSTEM AND METHOD TO UNIFORMLY ACCESS DEVICES

Inventors:

Koral Ilgun and Kedar Madineni

Prepared By:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard, 7th Floor
Los Angeles, California 90025-1026
(310) 207-3800

SYSTEM AND METHOD TO UNIFORMLY ACCESS DEVICES

Field of the Invention

[0001] The invention relates generally to computers and devices coupled with a computer, and, more specifically to providing a system and method for application programs to access devices in a uniform manner and for devices to access one another in a uniform manner.

Background of the Invention

[0002] Computers are machines that include various devices, whether they be included internally in a computer case, attached in a card cage, or attached externally. In general, application programs may access devices by communicating with device drivers in an operating system. Each kind of device of a plurality of a devices which may be coupled to a computer require a separate device driver. For an application program to communicate with each of the plurality of devices, the application program must include specialized software that allows the application to communicate with each of the plurality of kinds of devices and maintain information concerning the communications with each of the kinds of devices.

[0003] **Figure 1 (Prior Art)** illustrates a generalized conceptual architecture of prior art communication between application programs and devices in a computer system. Application programs 110 in user space 102 may communicate with various drivers 150 in an operating system kernel that exists in kernel space 104 to access a plurality of devices 160 at hardware level 106. To configure the collection of devices so that they function as desired within a system, an application program 110 accesses and controls the devices 160 by communicating with the appropriate device drivers 150. A drawback of this approach is that since the control requests are sent directly to the individual devices by the application program, the control requests have to be device specific. This hinders the application program from being device independent, as the application program must have the capability to communicate which each of the various devices it wishes to control. For example, for Application A to communicate with devices A, B and C, Application A must format three separate messages according

to the requirements of the three different device drivers. As such, Application A must send three separate messages to each of the drivers as shown by lines 120.

[0004] In some systems, each kind of device performs a unique function that may also be related to the function of one or more other devices. For example, a state change of one device may trigger an action on one or more other devices. This may result in one driver communicating with one or more other drivers, as shown by inter-driver communication 130. Although direct communication between devices is fast, such direct communication prevents a system from being designed and maintained modularly as each device is required to have knowledge of other related devices. That is, each device driver must know how to communicate with each of the other device drivers.

[0005] When, for example, functionality enhancements require the replacement or upgrade of a device, the replacement will require not only the modification or replacement of the device's driver, but also modification of control applications and/or other device drivers that rely on software features and control points of the device. That is, when a device is replaced or upgraded, each of the other devices and application programs that access the device must also be updated so as to be able to communicate with the replaced or upgraded device.

[0006] In many embedded computer systems, before executing a user-level request on a device, the application program is required to know the state of all related devices. This results in costly overhead in an embedded system, as each communication to a desired device may require multiple preliminary status requests of other related devices. In addition, when building a redundant device model, each device via its driver may be required to have intricate knowledge of related devices. As already discussed, this increases overhead due to the inter-device communication required. Moreover, hardware and software upgrades are difficult as they cause the re-writing of the device drivers of related devices.

[0007] In some systems, a request from an application program to device A may cause device A to generate another request to device B, which in turn may generate a further request to device C. If one of the nested requests fail, the operation needs to be rolled back and restarted. This requires drivers to keep complicated information about the states of other devices. Handling failures in this type of nested

request introduces great complexity in the drivers resulting from the dependency among devices.

[0008] In some systems, concurrent requests from multiple controlling application programs cannot be easily serialized and may lead to inconsistent states. For example, if Application A sends device-specific requests R1, R2, R3 and R4, and Application B sends device-specific requests S1 and S2, because of the scheduling mechanisms of the operating system, the requests may be received by the devices in an interleaved fashion such as, R1, S1, R2, S2, R3, R4. This may cause unintended results and may waste resources by causing errors that result in a roll back and retry.

[0009] When inter-device communication is needed in embedded systems, one drawback of allowing inter-device communication is that if one application's request to a driver is blocked due to the driver waiting for an I/O operation, a second request to the same device from another application may fail or block. In this circumstance, software may be added to each and every driver to return a "busy" indication to the second application. However, adding this software to each driver is burdensome and costly.

BRIEF DESCRIPTION OF THE DRAWINGS

[00010] The invention is illustrated by way of example and not by way of limitation in the figures of the accompanying drawings in which like references indicate similar elements. It should be noted that references to "an" or "one" embodiment in this disclosure are not necessarily to the same embodiment, and such references mean at least one.

[00011] **Figure 1 (Prior Art)** illustrates a generalized conceptual architecture of prior art communication between application programs and device in a computer system.

[00012] **Figure 2** illustrates a hardware environment in which one embodiment of the invention may execute.

[00013] **Figure 3** illustrates a generalized conceptual architecture of one embodiment of the invention.

[00014] **Figure 4** illustrates a conceptual architecture of one embodiment of the invention.

[00015] **Figures 5A** and **5B** illustrate a flow of actions taken pursuant to one embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

[00016] The system and method described herein are used to access and control related devices of different kinds in an embedded computer using a software interface referred to as a multiplexor (MUX) driver. The MUX driver allows for a uniform means of communication between one or more managing and/or controlling application programs and devices and among the devices themselves. The MUX driver presents a simpler view of the devices to application programs while providing a non-intrusive, transparent upgrade or replacement path for the individual devices and their driver software. Having a known interface for inter-device communication allows for easy replacement of devices and their drivers as no resulting reworking of other drivers is required.

[00017] **Figure 2** illustrates a hardware environment in which one embodiment of the invention may execute. System 200 may be an embedded computer system that includes a computing device 204, display monitor 222, and input devices such as a keyboard 216 and mouse 218. The display monitor and input devices are optional and are only needed if user interaction is required by application programs running on the computing device. The computing device 204 is coupled to a network 260 via communications interface 240. In one embodiment, network 260 is capable of communication using the User Datagram Protocol (UDP) and the Internet Protocol (IP). (For more information on UDP and IP, see J. Postel, User Datagram Protocol, RFC 768, August 28, 1980, *http://www.rfc-editor.org/rfc/rfc768.txt* and J. Postel, Internet Protocol, RFC 791, September 1981, *http://www.rfc-editor.org/rfc/rfc791.txt.*, incorporated herein by reference.) In other embodiments, the Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP) may be used in place of or in addition to UDP. (For more information on TCP and SCTP see T. Socolofsky, A TCP/IP Tutorial, RFC 1180, January 1991, *http://www.rfc-editor.org/rfc/rfc1180.txt*

and R. Stewart *et al.*, Stream Control Transmission Protocol, October 2000, *http://www.rfc-editor.org/rfc/rfc2960.txt.*, incorporated herein by reference.)

[00018] Computing device 204 may include a processor 210, memory 212, storage device 214, input/output (I/O) controller 220, display controller 224, and one or more devices 250. In one embodiment, devices 250 may include networking devices such as digital subscriber line (DSL) hardware, a Digital Signal Processor (DSP) device, a plain old telephone system (POTS) device, etc. The number of devices and kinds of devices are not limited. Some other devices include synchronous optical network (SONET) hardware, E1 hardware, T3 hardware, T1 hardware, asynchronous transfer mode (ATM) hardware, very high speed DSL (VDSL) hardware, Gigabit Ethernet hardware, fiber optic hardware, satellite transmission hardware, microwave communication hardware, infrared communication hardware, etc. In one embodiment, computing device 104 may include multiple processors.

[00019] In one embodiment, some or all of the methods which individually and/or collectively make up the invention described herein may be stored as software (*i.e.*, computer readable instructions) on storage device 214 and executed by processor 210 using memory 212. In another embodiment, the method may be stored as hardware or a combination of hardware and software such as firmware. In one embodiment, storage device 214 may be any machine readable medium, where a machine readable medium includes any mechanism that provides (*i.e.*, stores and/or transmits) information in a form readable by a machine (*e.g.*, a computer). For example, a machine readable medium includes read only memory (ROM); random access memory (RAM); magnetic disk storage media such as a hard disk drive or floppy disk drive; optical storage media such as a compact disk read-only memory (CD-ROM) and a readable and writeable compact disk (CD-RW); flash memory devices including stick and card memory devices; coupled to bus 230 via electrical, optical, acoustical or other form of propagated signals (*e.g.*, carrier waves, infrared signals, digital signals, etc.); etc. The various components of computing device 204 may be coupled to one another via bus 230. Bus 230 may be any well-known or proprietary bus. In addition, one or more buses may be included in various embodiments (*e.g.*, processor, memory and/or peripheral busses).

[00020] In one embodiment, computing device 204 may be a card connected to a back plane and each of devices 250 maybe separate cards connected to the same back plane. In this embodiment, system 200 may be a rack system or a card cage. In another embodiment, system 200 may include a server computer as computing device 204, and devices 250 may be coupled internally and/or externally with computing device 204. In one embodiment, system 200 may include a well known operating system such as, in one embodiment, Linux.

[00021] **Figure 3** illustrates a generalized conceptual architecture of one embodiment of the invention. In one embodiment, the methods of the invention may be thought of as residing in the kernel space 304, where the kernel space 304 resides between the user space 302 and hardware 306. In another embodiment, the methods of the invention may reside as software executed in user space 302. The kernel refers to the software of an operating system in the UNIX® family of operating systems and its variants, such as Linux. In one embodiment, the kernel is that of the Linux operating system. The system and method described herein are not limited to UNIX® derived operating systems, and may be used with any operating system whether the operating system includes conceptual architecture akin to a kernel space and a user space or not.

[00022] In one embodiment, one or more application programs such as applications 310 exist in user space 302. In one embodiment, the application programs 310 may exist on one computing device, computer or system. In another embodiment, application programs may exist on multiple computing devices, computers or systems. The application programs communicate with devices 360 via high-level MUX driver interface 330 of MUX driver 320 that exists in kernel space 304. In one embodiment, the communication method used between application programs 310 and MUX driver 320 may, in LINUX, be via /proc files, ioctl's or in other embodiments and/or other operating systems, other system calls or direct function calls. In other embodiments (not shown) where there is no kernel/user space separation, communication between application programs and the MUX driver may be via direct function call. In one embodiment, MUX driver 320 may be moved to and exist in user space 302; in such an embodiment, drivers 350 remain in kernel space 304 and devices 360 remain as hardware 306. The MUX driver 320 passes information between devices 360 and application programs 310. For example, the application programs may send device

control requests and status requests to MUX driver 320. In one embodiment, for each device or group of same devices, there is a driver 350 associated with the device or group of devices. In addition, each of the device drivers 350 may communicate with one another via a low-level MUX driver interface 340 provided by MUX driver 320. In one embodiment, low-level MUX driver interface 340 and high-level MUX driver interface 330 may be the same interface.

[00023] In one embodiment, the application programs may provide a user access from the user space 302 to the devices 360 at hardware level 306. MUX driver 320 allows application programs 310 from user space 302 to access devices 360 at hardware level 306 in a uniform manner by providing high-level MUX driver interface 330. In this way, the application programs are not required to conform to the unique communication requirements of each of the drivers 350. This makes communication and monitoring of devices by application programs easier to implement and achieve. The application programs send a high-level message to the MUX driver, which then converts the high-level message into low-level requests that are sent to one or more devices.

[00024] In various embodiments, the application programs 310 may send control requests to one or more devices via MUX driver 320 as high-level messages. These control requests may be used for various purposes, such as, for example, to set or change one or more options or features of a designated device, to power up a designated device, to power down a designated device, to restart a designated device, to upgrade software on a designated device, such as a programmable read only memory (PROM) upgrade or firmware upgrade, etc. In various embodiments, the application programs 310 may send status requests to one or more local devices via MUX driver 320. These status requests may be used by some application programs 310 to monitor the status of the device(s) and determine whether to automatically send a communication to or otherwise notify a system administrator, to automatically take load balancing actions, to automatically reconfigure one or more devices, to automatically restart or shut down one or more devices, etc. Other application programs 310 may be user driven and respond to system operator requests for status information. Based on the status information received, the user may then issue commands via an application program that communicates control requests to the

devices via the MUX driver to balance the load on particular devices, to reconfigure one or more devices, to restart or shut down one or more devices, etc.

[00025] According to this architecture, the writing of application programs is simplified because each of application programs 310 only need be able to communicate via the high-level interface to MUX driver 320. This removes the need for the application programs to know how to communicate with each of disparate device drivers 350. Only MUX driver 320 needs to know how to communicate with each of drivers 350.

[00026] Similarly, when a device receives a request, it may need to learn the status of other devices. The devices may communicate with each other in a uniform manner via low-level interface 340 by sending messages to MUX driver 320 in a low-level format prescribed by the MUX driver. In this way, the devices need only be able to communicate with the MUX driver rather than each of the disparate devices and their device drivers. This simplifies writing of device drivers and eases the path of upgrades and integrating updated and new devices.

[00027] In addition, when a device changes its state, be it in response to a control request from an application program, sensing network conditions, detecting changes in physical connection status or any other occurrence, the device may send a state change notice to other related devices. Rather than communicating directly with each of the other devices in a system, according to the methods described herein, the device need only send a single, low-level message to the MUX driver indicating the state change. The MUX driver has the necessary intelligence to notify the appropriate devices, and may also notify any interested application programs, of the change in state of the device.

[00028] **Figure 4** illustrates a conceptual architecture of one embodiment of the invention. In various embodiments, application programs may be used to control a particular device or a particular class of devices local to the system on which the application program resides, and may be used to monitor a device or monitor a particular class of devices local to the system on which the application program resides. In one embodiment, control program 410 may be used to control digital subscriber line (DSL) hardware 462 such as a DSL modem. The DSL modem may support one or more varieties of DSL technology such as, for example, asymmetric DSL (ADSL),

symmetric DSL (SDSL) and G.lite. A user may wish to send a control request via control program 410 to DSL hardware 462 to change settings on the DSL modem, to restart the DSL modem, etc. To do so, control program 410 must be written so as to be able to communicate with DSL MUX driver 420. In this embodiment, the application programs, such as control program 410 and monitor program 412, communicate with hardware devices such as DSL hardware 462 via DSL MUX driver 420. In one embodiment, DSL MUX 420 may receive control and/or monitor program requests from application programs in the form of a high-level message specifying a group of devices or all devices. In this embodiment, DSL MUX 420 then issues appropriate commands or messages to each of the devices via the appropriate device drivers. In one embodiment, for example, DSL driver 452 is used to control and communicate with DSL hardware 462, Digital Signal Processor (DSP) driver 454 is used to communicate with DSP device 464, plain old telephone system (POTS) driver 456 is used to communicate with POTS device 466, etc. The number of devices and kinds of devices are not limited. Some other devices include synchronous optical network (SONET) hardware, E1 hardware, T3 hardware, T1 hardware, asynchronous transfer mode (ATM) hardware, very high speed DSL (VDSL) hardware, Gigabit Ethernet hardware, and the like. Further devices may include, for example, other fiber optic hardware, satellite transmission hardware, microwave communication hardware, infrared communication hardware, etc.

[00029] In one embodiment, the DSL MUX communicates with the drivers via function calls and the application programs communicate with the DSL MUX via any of a variety of well-known techniques. These techniques include via netlink sockets, ioctls, function calls, /proc file system, etc. In these embodiments, not only can information be passed from application programs through the DSL MUX to devices, but information can be passed from the devices through the DSL MUX to the appropriate application program. In this embodiment, the DSL MUX serves as a multiplexor for information passing to multiple DSL devices from a single control application program and as a demultiplexor for information passing from multiple DSL and/or other devices to control and/or monitor application programs. In this embodiment, the control application program may be a specialized DSL control application program, and the monitor application program may be a specialized DSL monitor application program.

[00030] In one embodiment, the MUX driver may regularly gather a plurality of statistics regarding the plurality of devices. In one embodiment, this may be achieved by monitoring the contents of existing messages sent by the devices responsive to requests from application programs. In another embodiment, this may be achieved by the MUX driver explicitly sending regular and periodic status requests to those devices coupled with the MUX driver. In one embodiment, the MUX driver may periodically forward a message reporting the plurality of statistics regarding the plurality of devices to those application programs that subscribed to such a message and/or to those application programs that the MUX driver had determined should receive the statistics messages based on a prior control request sent regarding the device.

[00031] In another embodiment, the MUX driver may, using the statistics mentioned in the prior paragraph, determine which of the devices issued errors that exceed a threshold number of errors. In this way the determination is based on the statistics. If the MUX driver determines that the error threshold has been exceeded for a device, the MUX driver may, in various embodiments, send a shut-down message to the device and/or send a message to a monitor and/or a control application program.

[00032] In one embodiment, the MUX driver may maintain a database of configuration information regarding devices retrieved from the devices, particularly those devices that are primary devices. Primary devices are those devices of which there are other similar or same devices, such that the secondary devices may be used as back up devices for the primary devices. By maintaining configuration information for the primary devices, when the primary devices fail or are shut down for some reason, a secondary device may be configured to match the configuration of the down primary device. That is, the configuration information for the primary device is applied to a secondary device when the primary device is taken off-line.

[00033] **Figures 5A** and **5B** illustrate a flow of actions taken pursuant to one embodiment of the invention. The MUX driver receives a message, as shown in block 512. In one embodiment, the message may contain various fields that are in attribute, value, length format. In one embodiment, the message may be either a high-level message or a low level message. In one embodiment, a message may contain a message type and any data relevant to that message type. The content of the message, and, more specifically, the message type determines what occurs next. In one

embodiment, messages are received and then processed sequentially in the order in which they are received. The message is analyzed, and the kind of message is extracted, as shown in block 514.

[00034] If the message is a registration request from a device driver, then the MUX driver registers the device by adding a device driver identifier to a database maintained by the MUX driver, as shown in block 522. In one embodiment, a function call identifier may be included in the registration request such that the function call identifier is also added to the database to be used for future communication with the driver. The flow of actions then continues at block 512.

[00035] If the message is a registration request from a monitor application program, as shown in block 530, then the MUX driver registers the monitor application program, as shown in block 532. In this way, a monitor application program may subscribe to events from a specified device, device class, or group of devices to receive messages when events at the specified device occurs. The flow of actions then continues at block 512.

[00036] If the message is a state change notice from a device received via a device driver, as shown in block 540, a check is then made to determine whether any monitoring application programs have registered to receive events from the device, as shown in block 542. In one embodiment, this is achieved by reviewing the database. If one or more application programs monitor the device, then state change information is forwarded to those application programs that monitor the device, as shown in block 544. In one embodiment, application programs that monitor the device include monitor application programs that explicitly subscribe to events regarding specified device or devices, and, in one embodiment, may also include control application programs that sent control requests regarding a particular device or class of devices. In this way, the control application programs do not register with the MUX driver or subscribe to events, and the MUX driver transparently retains information about devices and control applications that sent control requests regarding a particular device or devices. If there are no application programs monitoring the device as determined in block 542, and after the state change information has been forwarded to any monitoring application programs in block 544, the state change notice is converted into one or more low-level requests to be directed to drivers for other devices, as shown in block 546.

[00037] If the message is a control request from a control application program specifying one or more devices or kinds of devices, as shown in block 550, a check is made to determine whether the device is registered with the MUX driver by checking the database, as shown in block 552. In one embodiment, the control request may be a status request for information concerning a particular device, a particular class of devices, or all devices. In another embodiment, the control request may be a command to one or more of the devices to change or otherwise update one or more specified settings, to restart, to shut down, etc. If the specified device or devices are accessible via the MUX driver, the control request is converted into one or more low-level requests to be sent to the specified device(s), as shown in block 554. In one embodiment, this communication is achieved via the function stored in the database indexed by the device identifier. If the device specified in the control request is not registered with the MUX driver, that is, it is not in the database, then the request is dropped and an error message is returned to the requesting control application program, as shown in block 556.

[00038] After the state change notice and the control request are converted into one or more low-level requests, as shown in blocks 546 and 554, the low-level requests are sent to one or more drivers, as shown in block 560. This can either be done in a sequential manner with a particular order or it can be done in parallel depending on the nature of the high-level request. If when done in a sequential manner any request in the sequence fails, the rest of the requests in the sequence are not executed. Responses to the low-level requests are then collected, as shown in block 562. The MUX driver then determines whether each of the particular low-level requests was successful by examining the collected results, as shown in block 564. If the low-level request was unsuccessful, then any previous low-level requests associated with the unsuccessful low-level request or otherwise related to the unsuccessful low-level request are rolled back, as shown in block 566. An example of an unsuccessful low-level request may be a control request that attempts to modify a setting or configuration which the particular device will not allow to be modified, such that the device rejects the corresponding resulting low-level request.

[00039] After determining that the low-level requests were successful in block 564 and after the roll back necessitated by one or more unsuccessful low-level requests

in block 566, an appropriate result code is sent to the initiating control application program or driver, as shown in block 568. Flow then continues at block 512.

[00040] In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes can be made thereto without departing from the broader spirit and scope of the invention as set forth in the appended claims. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.